

Speeding up Graph Algorithms via Switching Classes

Nathan Lindzey *

Colorado State University, Fort Collins CO 80521, USA

Abstract. Given a graph G , a *vertex switch* of $v \in V(G)$ results in a new graph where neighbors of v become nonneighbors and vice versa. This operation gives rise to an equivalence relation over the set of labeled digraphs on n vertices. The equivalence class of G with respect to the switching operation is commonly referred to as G 's *switching class*. The algebraic and combinatorial properties of switching classes have been studied in depth; however, they have not been studied as thoroughly from an algorithmic point of view. The intent of this work is to further investigate the algorithmic properties of switching classes. In particular, we show that switching classes can be used to asymptotically speed up several super-linear unweighted graph algorithms. The current techniques for speeding up graph algorithms are all somewhat involved insofar that they employ sophisticated pre-processing, data-structures, or use “word tricks” on the RAM model to achieve at most a $O(\log(n))$ speed up for sufficiently dense graphs. Our methods are much simpler and can result in super-polylogarithmic speedups. In particular, we achieve better bounds for diameter, transitive closure, bipartite maximum matching, and general maximum matching.

1 Introduction

The runtime of an algorithm is intimately related to how an instance is represented. Recall that the runtimes of the first generation of graph algorithms were expressed solely in terms of n , the number of vertices. This analysis was natural since at this time graphs were represented in $\Theta(n^2)$ space via their adjacency matrix. It was soon noticed that if $m = o(n^2)$, then a variety of graph algorithms could be sped up by first computing the adjacency list from the adjacency matrix, then running the algorithm on the more efficient adjacency list representation. This motivated the introduction of m to the runtime of graph algorithms and it is now customary in algorithm design to assume that a graph instance is given in the form of its adjacency list.

We introduce \tilde{m} as a measure of complexity and show many classical graph algorithms can be analyzed in terms of \tilde{m} . This is a significant measure of complexity since $\tilde{m} = O(m)$ but $\tilde{m} \neq \Theta(m)$. In particular, if $\tilde{m} = o(m)$, then several graph algorithms can be asymptotically sped up by computing the so-called *partially complemented adjacency list* (pc-list) from an adjacency list, then running the algorithm on the more efficient partially complemented adjacency list representation.

* lindzey@math.colostate.edu, Mathematics Department, Colorado State University, Fort Collins, CO, 80523-1873 U.S.A.

The pc-list [4] is a natural generalization of the adjacency list that involves an additional $O(n)$ bits of storage to represent *vertex switches*. When a vertex is switched, its neighbors become nonneighbors and nonneighbors become neighbors. A (di)graph afforded such a switching operation is commonly referred to as a *switching class* [17]. Figure 1 of the appendix demonstrates how a pc-list can represent a switching class which can in turn be used to obtain a more compact representation of a graph. Algebraic and combinatorial properties of switching classes have been studied in depth [17,2]; however, they have not been studied as thoroughly from an algorithmic point of view. The intent of this work is extend [4] by further investigating algorithmic properties of switching classes.

In [4] canonical $\Theta(n + m)$ unweighted graph algorithms were developed for switching classes; however, due to the linear-time solvability of these problems, the pc-list provided no asymptotic speed up in runtime. We extend this work by developing switching class algorithms for classical unweighted graph problems for which no linear-time algorithm is known. We show that for sufficiently dense graphs, the pc-list can provide super-polylogarithmic speed ups in runtime.

This is notable since the current techniques for speeding up algorithms over dense instances are all somewhat involved and achieve at most a $O(\log(n))$ speed up. A data-structure in [11] is given that allows one to work on the complement of a graph without constructing it; however, the algorithms they consider are linear and do not improve any of the results established in [4]. The techniques in [3] are notable in that they achieve a $O(\log(n))$ speed-up for several canonical graph problems over arbitrary dense graphs (assuming the RAM model). Clever but complicated preprocessing in [6] allows for an asymptotic speedup that is logarithmic in the density of the graph that is at most $O(\log(n))$.

Our approach is much simpler insofar that it involves only basic preprocessing of the graph and slight modifications to existing algorithms.

2 Preliminaries

All graphs are assumed to be finite, labeled, directed, unweighted, and simple unless stated otherwise, and let \mathcal{G} denote the class of all such graphs on n vertices. Let $V(G)$ and $E(G)$ denote vertex set and edge set of G respectively. Let $E(A, B)$ denote the set of edges that have exactly one endpoint in $A \subseteq V$ and exactly one endpoint in $B \subseteq V$. Let $G[X]$ denote the subgraph induced by the vertex set $X \subseteq V$. An *out-switch* (*in-switch*) of a vertex changes out-neighbors (in-neighbors) to non out-neighbors (in-neighbors) and vice versa. A *Seidel-switch* of a vertex in an undirected graph changes neighbors to nonneighbors and nonneighbors to neighbors. Performing an out-switch and in-switch on the same vertex of an undirected graph is equivalent to Seidel-switching that vertex. Let $\neg_v^+(G)$, $\neg_v^-(G)$, and $\neg_v(G)$ be the graphs obtained by out, in, and Seidel switching on a vertex $v \in V(G)$ respectively. It is easy to see that the order in which vertices are switched does not matter, so let $\neg_U^+(G)$, $\neg_U^-(G)$, and $\neg_U(G)$ be the graph obtained by out, in, and Seidel switching on a set $U \subseteq V(G)$. If a mixed sequence of in and out switches are permitted, then let $\neg_{I,O}^\pm(G)$ be the graph obtained by

Gale-Berlekamp switching where $I, O \subseteq V(G)$ are the subsets of vertices that have been in-switched and out-switched respectively.

Definition 1. Let $G, H \in \mathcal{G}$. Then $G \sim^* H$ iff $\exists U \subseteq V$ such that $\neg_U^*(G) \cong H$ with respect to some switching operation $*$.

Proposition 1. \sim^x is an equivalence relation over \mathcal{G} .

Definition 2. Let $\mathcal{C}_G^* = \{H \in \mathcal{G} : G \sim^x H\}$. Then \mathcal{C}_G^* is the switching class of G with respect to some switching operation $*$.

In particular, we let \mathcal{C}_G^+ , \mathcal{C}_G^- , \mathcal{C}_G^\pm , and \mathcal{C}_G denote the *in*, *out*, *Gale-Berlekamp*, and *Seidel switching class* of G respectively. It is worth noting that in-switching classes and out-switching classes have an algebraic structure similar to Gale-Berlekamp switching classes [16]. It is routine to show that \sim^+ and \sim^- form an equivalence relation over \mathcal{G} that gives rise to the Abelian group $\mathbb{Z}_2^{n^2-2n}$.

Definition 3. The *partially complemented adjacency list (pc-list)* of a graph G (with respect to some switching operation) is an adjacency list outfitted with a constant number of bitstrings of length n that represent vertex switches.

If a vertex v is switched, then we let $\tilde{N}(v)$ denote its doubly-linked neighborlist in the pc-list of \tilde{G} . If v is unswitched, then we let $N(v)$ denote the doubly-linked neighbor list of v in the pc-list of \tilde{G} . For any switched vertex v , we still refer to $\tilde{N}(v)$ as the neighborlist of v even though its elements are actually non-neighbors in the original graph.

Proposition 2. \overline{G} is the graph obtained by out-switching (in-switching) all of the vertices.

The proposition above is useful due to the fact that some graph classes can be recognized by considering properties of their complements [14]. Unfortunately, constructing the complement graph \overline{G} is an $\Omega(n^2)$ operation which precludes any linear-time bound. The pc-list has proved useful in this context since it represents \overline{G} implicitly which obviates the $\Omega(n^2)$ cost of constructing \overline{G} [4,14].

The pc-list was motivated by McConnell's *complement-equivalence classes* [13]. It is straightforward to see that the symmetric complement-equivalence classes of [4] coincide with Seidel switching classes and in-out complement-equivalence classes [4] coincide with Gale-Berlekamp switching classes. Due to this correspondence, it seems natural to couch the pc-list in terms of the existing theory of switching classes.

It is obvious that we should seek out small members of switching classes to obtain a more succinct representation of a given graph. Ideally, we should seek a member of a switching class with the fewest edges.

Definition 4. A *minimum representative* of a switching class \mathcal{C}_G^* is a not necessarily unique graph $\tilde{G} \in \mathcal{C}_G^*$ having minimum edge cardinality \tilde{m} .

If we limit ourselves to strictly out-switches or strictly in-switches, the following lemma shows that we can easily construct a minimum representative using a greedy algorithm.

Lemma 1. [4] *A minimum representative $\tilde{G} \in \mathcal{C}_G^+$ ($\tilde{G} \in \mathcal{C}_G^-$) can be constructed in $O(n + m)$ time.*

Proof. Visit each vertex and if switching it reduces the edge count, do so. For out switching, if there are more than $n/2$ elements in v 's neighbor list, switch v and replace the neighbor list with non-neighbors of v . The work for creating the list of non-neighbors can be charged to visiting the neighbors of v . For in switching, if v appears more than $n/2$ times in the adjacency list of G , then switch v . The work is clearly $O(m)$.

Observe that if both in-switches and out-switches are allowed, then the algorithm in the proof of Lemma 1 no longer guarantees that the representative is a minimum. This is because edges can reappear while constructing the representative. It is known that computing minimum representatives for Gale-Berlekamp and Seidel switching classes is NP-hard and is even hard to approximate within a constant factor of the optimum [16,9]. There do however exist randomized linear-time $(1 + \epsilon)$ -approximation schemes for computing a minimum representative $\tilde{G} \in \mathcal{C}_G^\pm$ [12]. This allows one to obtain a representative $G' \in \mathcal{C}_G^\pm$ such that $|G'| = \Theta(|\tilde{G}|)$ in $O(n \log(n) + m \log(n))$ time with high probability.

3 Basic Algorithms for Switching Classes

3.1 Traversal

Traversal algorithms for out-switching classes first appeared in [4] where the existence of $O(n + \tilde{m})$ algorithms for traversal on Seidel switching classes was left open. We show that these algorithms can be obtained in a straightforward manner through a slight modification of the pc-list data-structure and the traversal algorithms of [4,10]. We refer the reader to [4,10] for a more thorough treatment. We begin with an intuitive explanation as to why the pc-list is able to provide asymptotic savings in runtime for graph traversal.

Let \mathcal{A} be a graph traversal algorithm and assume there exists an oracle \mathcal{O} such that for any current vertex v , it returns in $O(1)$ time either an undiscovered neighbor v or reports that all of v 's neighbors have been discovered. If \mathcal{A} considers a vertex that has already been discovered, then we shall call this a *bad query*. It is clear that the runtime of \mathcal{A} with oracle \mathcal{O} is $\Theta(n)$ since \mathcal{A} can make no bad queries. This is no longer the case if we run \mathcal{A} without \mathcal{O} since we might have $\omega(n)$ bad queries for arbitrary graphs. In this case, the runtime of \mathcal{A} is dominated by bad queries since we could have as many as $O(m)$. However, if the size of G 's pc-list is asymptotically smaller than its adjacency list, then we can obtain a tighter upper bound on the number of bad queries that can occur during an execution of algorithm \mathcal{A} without use of an oracle. This is due to the fact that every bad query of BFS or DFS can be charged to an element of the pc-list data-structure [4,10].

Theorem 1. [4] Given $\tilde{G} \in \mathcal{C}_G^+$, BFS on G can be done $O(n + \tilde{m})$ time.

Theorem 2. [4,10] Given $\tilde{G} \in \mathcal{C}_G^+$, DFS on G can be done in $O(n + \tilde{m})$ time.

Proposition 3. Let $S \subseteq V$ be a set of Seidel switched vertices and let $H = G[S] \cup G[S - V]$. Then $\neg_S(G)$ is isomorphic to the graph $H' = (V(H), E(H) \cup \overline{E}(S, V - S))$ where $\overline{E}(S, V - S)$ is the complement of the cut induced by $(S, V - S)$.

Given an adjacency list representation of G and a set of Seidel switched vertices $S \subseteq V$ such that $|E(\neg_S(G))| < |E(G)|$, a pc-list data-structure that represents $\neg_S(G)$ can be constructed in $O(n + m)$ time as follows. Let v be an arbitrary vertex. If v is switched, set its bit to 1, add all of its neighbors in S and its nonneighbors in $V - S$ into its neighborlist. If v is unswitched, set its bit to 0, add all of its neighbors in $V - S$ and its nonneighbors in S into its neighborlist. Relabel the vertices so that the members of $V - S$ have a smaller label than members of S , then radix-sort the pc-list with respect to this new labeling. The sort has the effect of making all nonneighbors of v appear consecutively in v 's neighborlist. Finally, insert a dummy vertex between the two elements u and w of v 's neighborlist such that u is switched and w is unswitched.

The same algorithms given in [4] and [10] can be used on this pc-list representation with the following modification; once v 's dummy vertex is visited during a scan of its neighbor list, flip v 's bit in the pc-list. If v is switched, then the flip has the effect of treating elements after the dummy vertex as actual neighbors of v . If v is unswitched, then the flip has the effect of treating elements after the dummy vertex as nonneighbors of v . Accounting for this modification in the traversal algorithms of [4,10] is trivial. The foregoing gives the following result.

Theorem 3. Given $\tilde{G} \in \mathcal{C}_G$, BFS and DFS on G can be done in $O(n + \tilde{m})$ time.

3.2 Contraction

Another basic graph-theoretic operation is contraction of a subset of vertices. Henceforth we shall assume that all switching operations are out-switches for ease of exposition. Let $\tilde{G} \in \mathcal{C}_G^+$ be a minimum representative, let $n(B)$ be the number of vertices of a subset $B \subseteq V$, $\tilde{m}(B)$ be the number of edges incident to vertices of B in the graph \tilde{G} . The results of this section will be needed for computing maximum matchings of general graphs. Without loss of generality, we assume that the pc-list of G is sorted by vertex label.

Lemma 2. Given $\tilde{G} \in \mathcal{C}_G^+$, a set $B \subseteq V$ can be contracted to vertex \hat{b} in $O(n(B) + \tilde{m}(B))$ time.

Proof. To build the neighbor list of \hat{b} , we first build two doubly-linked neighbor lists X, Y that correspond to the contraction of all the switched vertices $S \subseteq B$ and unswitched vertices of $B - S$ respectively.

Initialize X to be $\tilde{N}(s)$ some $s \in S$ and let $b, c \in S$. Then contracting b and c together corresponds to taking the intersection $\tilde{N}(b) \cap \tilde{N}(c)$. Since the

neighborlists are sorted, taking the intersection $\bigcap_{b \in S} \tilde{N}(b)$ can be computed in $O(n(B) + \tilde{m}(B))$ using a routine similar to the merge routine of merge-sort as follows. Let $L[i]$ denote the i th element of a doubly-linked list L .

If $X[i] = \tilde{N}(b)[j]$, then the comparison can be charged to the j th edge of b 's neighbor list. If $X[i] > \tilde{N}(b)[j]$, then the comparison can be charged to the j th edge of b 's neighbor list. If $X[i] < \tilde{N}(b)[j]$, then $X[i]$ is removed from the doubly-linked list X . Removing a vertex from X happens at most $\tilde{m}(B)$ times since each deletion can be charged to an element of $\tilde{N}(s)$.

Initialize Y to be $N(v)$ for some $v \in B - S$ and let $b, c \in B - S$. Suppose that $b, c \in B - S$ are unswitched, then contracting b and c together corresponds taking the union $N(b) \cup N(c)$. The union $Y = \bigcup_{b \in B - S} N(b)$ can clearly be computed in $O(n(B) + \tilde{m}(B))$.

To combine X and Y it suffices to scan X and remove all elements from X that also exist in Y . This can be done using a routine similar to the merge routine. At the end of the routine, define $\tilde{N}(\hat{b})$ to be X . Since the sizes of X and Y are each $O(n(B) + \tilde{m}(B))$, it follows that $\tilde{N}(\hat{b})$ can be constructed in $O(n(B) + \tilde{m}(B))$ time.

Lemma 3. *Let β be a collection of vertex-disjoint subsets. Then the contracted graph \tilde{G}/β can be computed in $O(n + \tilde{m})$.*

Proof. By Lemma 2 we can perform all of the contractions in time $\sum_{B \in \beta} n(B) + \tilde{m}(B) = O(n + \tilde{m})$. After performing the contractions, the pc-list must be cleaned up so that vertices subsumed by contractions are no longer referenced. This can be done by radix-sorting and removing duplicates in $O(n + \tilde{m})$ time.

4 Super-Linear Graph Algorithms

In this section, we show that given a graph $G = (V, E)$, spending $O(n + m)$ time to compute an $O(n + \tilde{m})$ space pc-list representation of G gives rise to better bounds for several canonical unweighted graph algorithms.

4.1 Diameter and Transitive Closure

At present, the most efficient combinatorial algorithm (ignoring log factors) for computing the diameter and transitive closure of a graph to our knowledge is the naive $O(n^2 + nm)$ algorithm, that is, calling BFS from each vertex. The following results are straightforward.

Theorem 4. *The diameter of a graph G can be computed in $O(n^2 + n\tilde{m})$ time.*

Theorem 5. *The transitive closure of a graph G can be computed in $O(n^2 + n\tilde{m})$ time.*

Proof. Compute \tilde{G} in $O(n + m)$ time, then run the naive algorithm from each vertex using the BFS algorithm of [4]. The runtime of this algorithm is $O((n + m) + n(n + \tilde{m})) = O(n^2 + n\tilde{m})$.

Since $n^2 + n\tilde{m}$ is never worse than $n^2 + nm$ and is sometimes better, this is indeed a better bound for both diameter and transitive closure.

At this point it is natural to ask when a graph *benefits* from its pc-list representation, that is, when its pc-list representation is asymptotically smaller than its adjacency-list representation. It is easy to see that there are at least twice as many graphs that benefit from pc-lists as there are graphs that benefit from adjacency-lists. This is because the pc-list is a generalization of the adjacency-list and the complement of any graph whose adjacency-list representation is asymptotically smaller than its adjacency-matrix representation must have a minimum representative $\tilde{G} \in \mathcal{C}_G^+$ such that $\tilde{m} = o(m)$. For instance, the class of graphs whose complement is sparse benefits from its pc-list representation simply by out-switching all of its vertices.

It would be interesting to give precise conditions for when a graph benefits from its pc-list representation, but for now, our intuition tells us that very dense graphs and “unbalanced” graphs (graphs with vanishingly few vertices of average valency) appear to benefit most from the pc-list representation.¹ On the other hand, since almost all graphs are roughly $n/2$ -regular, the pc-list does not provide an asymptotically smaller representation for most graphs. The techniques of [3] and [6] are notable since they give logarithmic speedups in runtime for almost all graphs.

4.2 Hopcroft-Karp Bipartite Maximum Matching

Notice that switching in general does not preserve the bipartite property. Given a bipartite graph G , define the *bipartite switching class* of G (with respect to some switching operator), such that neighbors of v become nonneighbors and nonneighbors of v that lie outside of v ’s partition class become neighbors. In this section, all switches are assumed to be bipartite out-switches. Familiarity with the bipartite maximum matching problem is assumed.

The Hopcroft-Karp algorithm consists of *phases*, each of which strictly increases the size of a current matching. In [8] it is shown that a phase consists of a call to a modified BFS routine followed by a call to a modified DFS and that only $O(\sqrt{n})$ phases are needed to compute a maximum matching. These routines can be implemented to run in $O(n+m)$ time which gives rise to a $O(\sqrt{n}(n+m))$ bound for computing a maximum matching of a bipartite graph.

The purpose of running the modified BFS is to discover a directed acyclic level graph L such that any path connecting two unmatched vertices in L corresponds to a shortest augmenting path in G . A modified DFS is then conducted on L in order to find a maximal set of vertex-disjoint augmenting paths \mathcal{P} in G . These steps are repeated until a phase is encountered such that $\mathcal{P} = \emptyset$, in which case the matching M must be maximum by a theorem of Berge.

It suffices to show that given \tilde{G} , a phase of Hopcroft-Karp can be implemented to run in $O(n + \tilde{m})$ time. Let BFS* and DFS* be pc-list implementations of the aforementioned modified BFS and DFS routines. The BFS* routine is essentially

¹ Perhaps it is possible to make this intuition well-defined via discrepancy theory.

the same as the BFS algorithm of [4] except for the following modifications. Let $level(v)$ denote the BFS level of a vertex v .

1. The undiscovered vertices are divided into two doubly-linked lists U_A and U_B . If the current vertex $v \in A$, then BFS^* only considers vertices in U_B (similarly for $v \in B$).
2. The discovered vertices are kept in an array of doubly-linked lists \mathcal{L} that represents a partition of the discovered vertices by BFS level. In particular, a discovered vertex v resides in the doubly-linked list at index $level(v)$ of \mathcal{L} . This array represents the levels in the DAG L .
3. Let k be the level of the first unmatched vertex in B that has been dequeued. Once all vertices at level k have been dequeued, the routine returns \mathcal{L} .

It is clear that the aforementioned modifications to the BFS routine of [4] can be accomplished in $O(n + \tilde{m})$ time. Pseudocode of BFS^* that achieves this bound is given in the appendix.

The vertices of \mathcal{L} form the initial set of undiscovered vertices for the modified DFS routine. Notice that explicitly constructing the level DAG L from \mathcal{L} would exceed the $O(n + \tilde{m})$ time bound; however, this is unnecessary since \mathcal{L} provides an implicit representation of L . A precondition to the DFS algorithm of [10] is that doubly-linked list of undiscovered vertices is ordered according to the ordering of the vertices of the pc-list. For each $i \in \{1 \dots k\}$, the ordering of $\mathcal{L}[i]$ might not respect the ordering of the vertices in the sorted pc-list; however, this can be corrected by performing a radix-sort on \mathcal{L} in $O(n)$ time. The DFS^* routine is the same as the DFS algorithm in [10] except for the following modifications.

1. If a vertex $v \in \mathcal{L}[level(v)]$ is current, then U points to the doubly-linked list $\mathcal{L}[level(v)+1]$ so that v only considers undiscovered neighbors at $level(v)+1$.
2. When a path P from the designated source vertex s to an unmatched vertex in B has been found, the routine adds $P - s$ to the set of vertex-disjoint augmenting paths and restarts DFS^* from s .

The vertices of $P - s$ cannot be considered in subsequent DFS^* searches since they are removed from \mathcal{L} once they are discovered. Since the union of vertex-disjoint paths \mathcal{P} has size $O(n)$, it is clear that DFS^* can be implemented to run in $O(n + \tilde{m})$ time. Pseudocode of DFS^* that achieves this bound is given in the appendix.

Finally, since a set of matched edges M has size $O(n)$, it is clear that the symmetric difference between M and the edges of the paths in \mathcal{P} can be computed in $O(n)$ time. This completes the description of a phase. Because BFS^* and DFS^* are $O(n + \tilde{m})$ and updating a matching takes $O(n)$ time, we obtain the following result.

Lemma 4. *A phase of Hopcroft-Karp can be implemented in $O(n + \tilde{m})$ time.*

By Lemma 4 and the fact that only $O(\sqrt{n})$ phases are needed to compute a maximum matching of a bipartite graph, we obtain the following result.

Theorem 6. *A maximum matching of a bipartite graph G can be computed in $O(n^{1.5} + \sqrt{n}\tilde{m} + m)$ time.*

Pseudocode for Hopcroft-Karp that achieves the bound above is given in the appendix. In [1] it is shown that finding maximum matchings of complement biclique graphs can be used as a heuristic to enumerate large cliques in graphs. It is possible that the pc-list could make this heuristic more efficient in practice by circumventing the construction of the complement.

4.3 Gabow-Tarjan Maximum Matching

A full discussion of the Gabow-Tarjan maximum cardinality matching algorithm [7] is beyond the scope of this paper. The following is only a rough sketch of the algorithm. Familiarity with the maximum matching problem is assumed.

It is well known that the maximum matching problem for general graphs is complicated by the existence of alternating odd cycles (blossoms) [5]. The Gabow-Tarjan maximum matching algorithm consists of $\lceil \sqrt{n} \rceil$ iterations of so-called *phase 1* (not to be confused with Hopcroft-Karp's phases) followed by $O(\sqrt{n})$ calls to *find_ap_set* [7]. Phase 1 consists of running *find_ap_set* on a contracted graph G/β and expanding unweighted blossoms afterwards. The *find_ap_set* routine is a depth-first based search responsible for discovering augmenting paths and blossoms of the input graph. A pre-condition to *find_ap_set* is that the input graph G/β is contracted with respect to a set of blossoms β . Once *find_ap_set* halts, the routine returns a maximal set of vertex-disjoint augment paths in the contracted graph. These paths in the contracted graph are translated into vertex-disjoint augmenting paths in the original graph in $O(n)$ time and the current matching is updated with respect to those paths. In addition to finding these paths, an execution discovers a set of new blossoms as well as a set of previously discovered blossoms to be expanded. This gives rise to a new set of blossoms to be contracted before the next call to *find_ap_set*. A blossom is expanded if its dual variable z becomes non-positive, which establishes an invariant that contracted blossoms \hat{b} always have positive weight after the first execution of *find_ap_set*. Unweighted blossoms are expanded because they might be “hiding augmenting paths” [5]. Expanding these blossoms gives *find_ap_set* a chance to find these hidden augmenting paths in the next iteration. The maximum cardinality case is treated as a special case of their minimum weight matching algorithm and is shown to run in time $O(\sqrt{n}(n + m))$ since the algorithm performs $O(\sqrt{n})$ iterations of *find_ap_set* which runs in time $O(n + m)$. In [7] it is shown that blossom, augmenting path, and dual variable maintenance all take $O(n)$ time and space during an iteration of phase 1.

Lemma 3 shows that we can build the contracted graph in time proportional to \tilde{G} , so it remains to show that *find_ap_set* can be conducted in $O(n + \tilde{m})$ time. We shall assume that we have obtained a minimum member \tilde{G} of G 's out-switching class. Recall that a vertex of \tilde{G} is switched if and only if it has $n/2$ or more neighbors in the original graph. It follows that we can build an adjacency

lookup vector $\mathbf{v}[]$ for each switched vertex v in $O(n + m)$ time. This lookup vector will allow us to answer if some vertex u is adjacent to a switched vertex v in the original graph in $O(1)$ time.

Lemma 5. *The $find_ap_set$ routine can be implemented in $O(n + \tilde{m})$ time.*

It suffices to show that the subroutine $find_ap$ can be implemented in $O(n + \tilde{m})$ time [7]. Let $b(v)$ denote the blossom that v currently belongs to. It is important to note that $find_ap$ only performs grow steps and blossom steps. We show how to perform each of these steps when the current vertex is switched. The following invariants will be needed to simplify the analysis of our modification to $find_ap$.

1. Let x be the current vertex. If an edge xy is scanned and $b(y)$ became outer after $b(x)$, then a blossom step is performed and every vertex in $b(y)$ has been completely scanned [7].
2. The order in which vertices become outer is given by the ordering of a doubly-linked list OUT .
3. If the current vertex performs a blossom step, then no more grow steps are possible from the current vertex.

The second and third invariants are not part of the specification of their algorithm; however, it can be implemented to maintain these invariants without affecting the correctness or resource bounds of the algorithm.

It is straightforward to modify the DFS algorithm of [10] so that when a current outer vertex u is considering an inner undiscovered neighbor v , the next outer current vertex becomes v' where $vv' \in M$ is a matched edge. The details of the blossom step are slightly more involved. It helps to view the blossom steps as DFS where the “undiscovered vertices” are those vertices that have an outer label that is greater than the current vertex’s outer label. Once a blossom step is performed, the inner vertices along that blossom become current (and therefore outer) in order of their DFS discovery time and the routine proceeds recursively.

Assume that a current vertex x has no more grow steps, that is, there are no more undiscovered vertices reachable from x . If x is not switched, then proceed as usual in $find_ap$; otherwise, assume that x is switched. The invariants guarantee that the blossom steps from x can be conducted by performing DFS where the “undiscovered vertices” are those vertices to the right of x in OUT . Let us refer to this ordered set of vertices as $OUT_{>x}$. The main obstacle is that we cannot use the DFS routine of [10] immediately to solve this problem since the ordering of the pc-list does not respect the ordering of $OUT_{>x}$. For this reason we must use $\mathbf{x}[]$ to determine adjacency in $O(1)$ time.

Let $y \in OUT_{>x}$. If $\mathbf{x}[y] = 1$, then a blossom step is performed. When a blossom step is performed, then $b(y)$ is removed from OUT since invariant 1 implies that these vertices have already been completely scanned and cannot be further used to discover an augmenting path. Assuming the blossom data-structures in [7], it follows that blossom steps take $O(n + \tilde{m})$ time.

If $\mathbf{x}[y] = 0$, then no blossom step is performed, but we can charge the lookup to y ’s entry in $\tilde{N}(x)$. But as in the routines of [4] and [10], we must guarantee that

x considers y only $O(1)$ times throughout the entire DFS execution on $OUT_{>x}$. To ensure this, each time we encounter a y such that $\mathbf{x}[y] = 0$, we add it to the end of an auxiliary doubly-linked neighborlist for $\tilde{N}'(x)$. This has the effect of building an ordered neighborlist for x that respects the ordering of OUT . Using this neighborlist, we can follow the restart step of the DFS algorithm in [10] to correctly and efficiently resume DFS on $OUT_{>x}$ when x returns from a recursive call. As in the bipartite case, keeping track of augmenting paths and updating the current matching can be accomplished in $O(n)$ time. Since *find_ap_set* can be implemented in $O(n + \tilde{m})$ time, we obtain the following result.

Theorem 7. *A maximum matching of a graph G can be computed in $O(n^{1.5} + \sqrt{n\tilde{m}} + m)$ time.*

5 Conclusions

We have demonstrated that switching classes can be used to obtain asymptotically better bounds for several graph algorithms through use of the pc-list data-structure. These improvements on algorithm resource bounds suggest that the pc-list is a more efficient data-structure than the adjacency list for several unweighted graph problems. But like any graph representation, it has its trade-offs. For instance, finding an arbitrary neighbor of a switched vertex v in the original graph takes $\Theta(|N(v)|)$ time whereas finding an arbitrary neighbor of an unswitched vertex takes $\Theta(1)$ time.

It may be tempting to believe that any unweighted graph algorithm can be implemented to work with a pc-list representation; however, it seems unlikely that the maximum matching algorithm of [15] is amenable to pc-lists. This is due to the fact that their approach requires a number of edges (so-called bridges) to be queued and processed at a later point in the execution of the algorithm. When a bridge is processed, it does not always progress the algorithm, that is, it may not lead to a grow step, produce an augmenting path, or discover a blossom. In light of this, processing a bridge cannot always be charged to a vertex or edge of \tilde{G} . We were unable to obtain a bound tighter than $O(m)$ for the number of bridges queued throughout the algorithm; however, it might be possible to modify the algorithm so that only bridges that lead to an augmenting path or blossom are considered.

In fact, there are classical super-linear graph problems that inherently cannot benefit from the pc-list. Two such examples are finding an Eulerian tour of G and computing the ear decomposition of G . Any algorithm for these problem must spend $\Omega(m)$ time for arbitrary graphs since the size of the output is proportional to the number of edges.

An obvious line of future work would be towards a more formal characterization of graphs that benefit from pc-list representations as well as the development of more pc-list algorithms. We conclude by thanking Ross M. McConnell for his insightful comments.

References

1. E. Balas and W. Niehaus. Finding large cliques in arbitrary graphs by bipartite matching. In Davis S. Johnson and Michael A. Trick, editors, *Cliques, Colouring, and Satisfiability, Second DIMACS Implementations Challenge, Oct. 11-13, 1993*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 29–52. American Mathematical Society, 1996.
2. Ying Cheng and Albert L Wells Jr. Switching classes of directed graphs. *Journal of Combinatorial Theory, Series B*, 40(2):169 – 186, 1986.
3. Joseph Cheriyan and Kurt Mehlhorn. Algorithms for dense graphs and networks on the random access computer. *Algorithmica*, 15(6):521–549, 1996.
4. Elias Dahlhaus, Jens Gustedt, and Ross M. McConnell. Partially complemented representations of digraphs. *Discrete Mathematics & Theoretical Computer Science*, 5(1):147–168, 2002.
5. Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, February 1965.
6. Tomás Feder and Rajeev Motwani. Clique partitions, graph compression and speeding-up algorithms. *J. Comput. Syst. Sci.*, 51(2):261–272, 1995.
7. Harold N. Gabow and Robert Endre Tarjan. Faster scaling algorithms for general graph-matching problems. *J. ACM*, 38(4):815–853, 1991.
8. John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.
9. Eva Jelínková, Ondrej Suchý, Petr Hliněný, and Jan Kratochvíl. Parameterized problems related to seidel’s switching. *Discrete Mathematics & Theoretical Computer Science*, 13(2):19–44, 2011.
10. Benson Joeris, Nathan Lindzey, Ross M. McConnell, and Nissa Osheim. Simple dfs for partially complemented digraphs. *Inf. Process. Lett. (submitted and available on arxiv.org)*, 2013.
11. Ming-Yang Kao, Neill Occhiogrosso, and Shang-Hua Teng. Simple and efficient graph compression schemes for dense and complement graphs. *J. Comb. Optim.*, 2(4):351–359, 1998.
12. Marek Karpinski and Warren Schudy. Linear time approximation schemes for the gale-berlekamp game and related minimization problems. In *STOC*, pages 313–322, 2009.
13. Ross M. McConnell. Complement-equivalence classes on graphs. In *Structures in Logic and Computer Science*, pages 174–191, 1997.
14. Ross M. McConnell and Jeremy Spinrad. Modular decomposition and transitive orientation. *Discrete Mathematics*, 201(1-3):189–241, 1999.
15. Silvio Micali and Vijay V. Vazirani. An $o(\sqrt{n})$ algorithm for finding maximum matching in general graphs. In *FOCS*, pages 17–27, 1980.
16. Ron M. Roth and Krishnamurthy Viswanathan. On the hardness of decoding the gale-berlekamp code. *IEEE Transactions on Information Theory*, 54(3):1050–1060, 2008.
17. J. J. Seidel. A survey of two-graphs. In *Colloquio Internazionale sulle Teorie Combinatorie*, pages 481–511, 1976.

6 Appendix

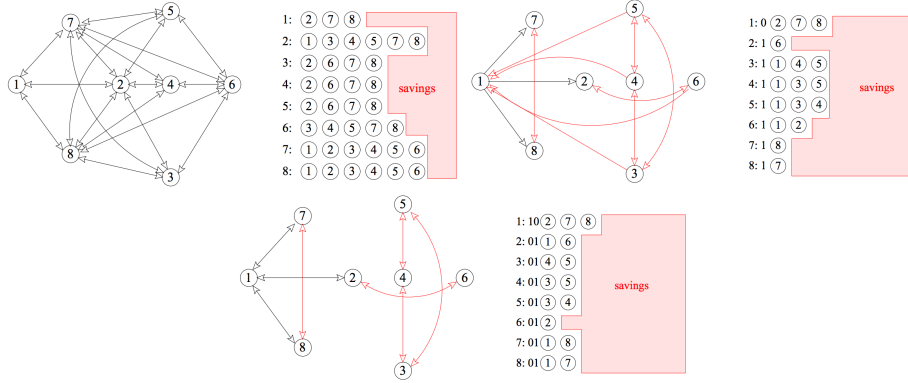


Fig. 1: An illustration of the partially complemented adjacency list. Setting G as the graph given in the leftmost picture, it is not hard to see that the graph obtained by out switching (the rightmost picture) and Gale-Berlekamp switching (the lower picture) are minimum representatives of \mathcal{C}_G^+ and \mathcal{C}_G^\pm respectively. This demonstrates that \overline{G} is not necessarily a minimum representative of G .

Algorithm 1: Hopcroft-Karp(G)

Data: A bipartite graph $G = (A, B, E)$

Result: A maximum matching M of G

```

1  $M \leftarrow \emptyset$ ;
2 Let  $\tilde{G}'$  be the pc-list of a minimum representative of  $\mathcal{C}_G^+$ ;
3 repeat
4   Let  $\tilde{G}$  be a copy of the pc-list  $\tilde{G}'$ ;
5    $\mathcal{P} \leftarrow \emptyset$ ;
6    $U_A, U_B \leftarrow A, B$ ;
7   connect  $s$  to each unmatched vertex in  $A$ ;
8    $\mathcal{L} \leftarrow \text{BFS}^*(s)$ ;
9    $\mathcal{L} \leftarrow \text{radix-sort}(\mathcal{L})$ ;
10  DFS*( $s$ );
11   $M \leftarrow \mathcal{P} \oplus M$ ;
12 until  $\mathcal{P} = \emptyset$ ;
13 return  $M$ ;

```

Algorithm 2: BFS*(s)

Data: A pc-list \tilde{G} , two global ordered doubly-linked lists U_A , U_B of the undiscovered vertices of A and B respectively.

Result: A partition \mathcal{L} that represents a level graph.

```

1 Let  $\mathcal{L}$  be an array of doubly-linked lists;
2 Let  $Q$  be a queue;
3  $k \leftarrow +\infty$ ;
4 enqueue( $s, Q$ );
5 repeat
6    $v \leftarrow \text{dequeue}(Q)$ ;
7   if  $\text{level}(v) > k$  then
8     break;
9   else if  $v \in A$  then
10    Let  $U$  point to  $U_B$ ;
11   else
12     if  $v$  is unmatched then
13        $k \leftarrow \text{level}(v)$ ;
14     Let  $U$  point to  $U_A$ ;
15   if  $v$  is unswitched then
16     for  $u \in N(v)$  do
17       if  $u$  is undiscovered then
18         remove( $u, U$ );
19         enqueue( $u, Q$ );
20         append( $u, \mathcal{L}[\text{level}(v) + 1]$ );
21   else
22     for  $u \in \tilde{N}(v)$  do
23       mark( $u, U$ );
24     for  $u \in U$  do
25       if  $u$  is unmarked then
26         remove( $u, U$ );
27         enqueue( $u, Q$ );
28         append( $u, \mathcal{L}[\text{level}(v) + 1]$ );
29 until  $Q$  is empty;
30 return  $\mathcal{L}$ 

```

Algorithm 3: DFS*(v)

Data: A pc-list \tilde{G} , a current undiscovered vertex v , a global array \mathcal{L} of ordered doubly-linked lists of undiscovered vertices, and global set of vertex-disjoint shortest augmenting paths \mathcal{P} .

Result: Discovers a maximal set of vertex-disjoint augmenting paths \mathcal{P} .

```

1  Let  $U = \mathcal{L}[\text{level}(v) + 1]$ ;
2  if  $v \neq s$  then
3    remove( $\mathcal{L}[\text{level}(v)], v$ );
4  if  $v \in B$  and  $v$  is unmatched then
5    Let  $P$  be vertices of the DFS stack;
6     $\mathcal{P} \leftarrow \mathcal{P} \cup \{P - s\}$ ;
7    DFS*( $s$ );
8  if  $v$  is unswitched then
9    for  $u \in N(v)$  do
10     DFS*( $u$ );
11 else
12    $u_v \leftarrow \text{head}(U)$ ;
13    $n_v \leftarrow \text{head}(\tilde{N}(v))$ ;
14   while  $u_v \neq \text{null}$  do
15     if  $u_v = n_v$  then
16        $u_v \leftarrow \text{next}(U, u_v)$ ;
17        $n_v \leftarrow \text{next}(\tilde{N}(v), n_v)$ ;
18     else if  $u_v > n_v$  then
19        $s \leftarrow n_v$ ;
20        $n_v \leftarrow \text{next}(\tilde{N}(v), n_v)$ ;
21       remove( $\tilde{N}(v), s$ );
22     else
23       DFS( $u_v$ );
24       // restarting step ...
25        $w = \text{prev}(\tilde{N}(v), n_v)$ ;
26       while  $w \neq \text{null}$  and  $w \notin U$  do
27          $t = w$ ;
28          $w \leftarrow \text{prev}(\tilde{N}(v), t)$ ;
29         remove( $\tilde{N}(v), t$ );
30       if  $w = \text{null}$  then
31          $u_v \leftarrow \text{head}(U)$ ;
32       else
33          $u_v \leftarrow \text{next}(U, w)$ ;

```

Algorithm 4: $\text{contract}(\beta)$

```

1 for  $B_i \in \beta$  do
2    $X, Y \leftarrow \emptyset$ ;
3   foreach  $b \in B_i = \{b_0, b_1, \dots, b_k\}$  do
4     if  $b$  is switched then
5       if  $X = \emptyset$  then
6          $X \leftarrow \tilde{N}(b)$ ;
7       else
8          $X \leftarrow X \cap \tilde{N}(b)$ ;
9     else
10       $Y \leftarrow Y \cup N(b)$ ;
11    $\tilde{N}(\hat{b}_i) \leftarrow X \oplus Y$ ;
12 Relabel each vertex  $v \in B_i$  with  $\hat{b}_i$ ;
13  $\tilde{G} \leftarrow \text{radix-sort}(\tilde{G})$ ;
14  $\tilde{G} \leftarrow \text{remove-duplicates}(\tilde{G})$ ;

```

Algorithm 5: $\text{Gabow-Tarjan}(G)$

```

1 Let  $\tilde{G}$  be the pc-list of a minimum representative of  $\mathcal{C}_G^+$ ;
2 Let  $T$  be the blossom tree of  $\tilde{G}$ ;
3  $i \leftarrow 0$ ;
4  $\beta \leftarrow \emptyset$ ;
5  $\tilde{G}' \leftarrow \tilde{G}$ ;
6 repeat
7   // phase 1
8    $\tilde{G} \leftarrow \text{contract}(\beta, \tilde{G}')$ ;
9    $\mathcal{P}, T \leftarrow \text{find\_ap\_set}(\tilde{G})$ ;
10  // search performs dual adjustments & expands nonpositive blossoms
11   $\beta \leftarrow \text{search}(\tilde{G}, T)$ ;
12   $M \leftarrow \mathcal{P} \oplus M$ ;
13   $i \leftarrow i + 1$ ;
14 until  $i > \lceil \sqrt{n} \rceil$ ;
15 repeat
16    $\mathcal{P} \leftarrow \text{find\_ap\_set}(\tilde{G})$ ;
17    $M \leftarrow M \oplus \mathcal{P}$ ;
18 until  $\mathcal{P} = \emptyset$ ;
19 return  $M$ ;

```
